Amendment t the Specification

Please replace the paragraph beginning on page 25, line 1, with the rewritten paragraph below.

Fig. 8 illustrates a Swing-type API 74 (region enclosed within the dotted lines). In Fig. 8, Button 30 is derived from the Component class 32. Also, the Peer mechanism links the Button object 30 with the graphics resources used to create and manipulate its graphic image 26, which is contained within Frame 9041. However, in this case, the JButtonPeer 82 (and the JComponentPeer 84, from which it is derived) are lightweight Peers, written entirely in Java and having no dependence on the windowing resources of the operating system. Instead, the JComponentPeer 84 intercepts key query commands related to the location and event status of the Button object 30 and routes them to a JButtonProxy object 88. In the present embodiment, the following methods of the Swing object are intercepted and redirected by the proxy object:

isShowing();
getLocationOnScreen();
getGraphics();
proxyRequestFocus();
requestFocus();
getInputContext();
getLocale();
nextFocus();
createImage();
getParent();

As a result, the JButtonProxy 88 responds to key events, such as focus, mouse, sizing, etc., directed to the original AWT-based control. JButtonProxy 88 is a subclass of the Swing JComponent class and, indirectly, of the JComponent class, and inherits methods and properties thereof. However, Java allows for customization (commonly referred to as "overriding") of inherited methods. In particular, the getParent() method of the JButtonProxy identifies as its parent the Frame ~~90~~41 of the target Button 30. Thus, the JButton "thinks" it belongs in the target's Frame, while the Frame knows nothing of the existence of the Swing component. Furthermore, when the Button 30 is asked to draw itself, the call to the drawing method of legacy AWT object is redirected by lightweight Peer 84 to JButtonProxy 88, which accesses the

drawing methods of the JButton class 48. This invokes the platf rm-ind pendent meth ds of Swing to paint the region of the screen originally allocated for the Button 30 in the AWT-based application.

Please replace the paragraph beginning on page 39, line 6, with the rewritten paragraph below.

Normally, Swing components are created only during the construction of the Peer object. However, in order to reproduce the behavior of the legacy AWT-based TextField control 170, JTextFieldPeer 172 must dynamically switch between the two Swing objects when the mode of use changes. This is accomplished by creating a new JTextFieldProxy 174 or JPasswordTextFieldProxy 176 object, depending on the settings of the echo character, and transferring the properties from the old to the new component. The appropriate Swing component is in effect "switched in" as a functional replacement for the AWT TextField, according to its mode of use. For example, if a legacy application is using ~~a~~ an AWT TextField control without password protection, no echo character is set. In this case, the AWTSwing JTextFieldPeer uses the Swing TextField component. Now, if the legacy application activates password protection, the AWT TextField will be assigned an echo character. When this occurs, the JTextFieldPeer creates an instance of Swing's JPasswordField component and substitutes it for the original AWT TextField control. This is done dynamically, so that each time the application changes the echo character status, the appropriate Swing replacement object (JTextField or JPasswordTextField) is created and used to replace the previous replacement object. Thus, the mode-switching capability of AWTSwing permits two Swing components to alternate as replacements for an AWT TextField component, depending on the manner in which the AWT TextField is being used by the application.

Please replace the paragraph beginning on page 39, line 26, with the rewritten paragraph below.

A flowchart representing the logic for dynamic TextField mode switching is presented in Fig. 18. Upon entry 192, the algorithm tests 178 the status of the echo character. If there has been no change in the state of the echo character, then whichever Swing control is currently in use (either JTextField or JPasswordField) is retained, and nothing needs to be done 190. On the other hand, if the echo status has changed, then it is necessary to swap the JTextField Swing component for a JPasswordField Swing component, or vice-versa. Before making this switch, the state (color, position, text, etc) of the control that is presently being displayed is saved 180. Once this information is captured, the current object can be destroyed 182, and its alternate created 184. After it is initialized 186, the previously saved state is applied t the ~~new~~ newly created Swing component, and the procedure is finished 190.

Please replace the paragraph beginning on page 40, line 20, with the rewritten paragraph below.

In contrast to this practice, it is normal for controls in Swing-based Java applications to receive their background color assignment from global look and feel settings in Swing (if it is not explicitly declared). Consequently, when upgrading from AWT to AWTSwing, the color inheritance mechanism must be modified to allow the background and foreground color of controls to be set by Swing, while also preserving the capability for controls to inherit this setting from their parent in legacy applications.

Please replace the paragraph beginning on page 41, line 1, with the rewritten paragraph below.

In yet another embodiment of the system and method disclosed herein, this is accomplished by adopting the following color inheritance order:

1. If the background color for the control is explicitly declared, AWTSwing uses this setting to display the control.

2. If the background color for the control is not explicitly declared, ~~and~~ AWTSwing attempts to get the color from the Swing settings.

3. If the background color for the control is not explicitly declared, and is not available from Swing settings, AWTSwing displays the control using the background color of the control's parent.

This inheritance scheme allows normal inheritance of background color from the global look and feel settings in Swing. However, it defaults to the AWT scheme of inheritance from the parent when these global settings are unavailable. An explicit declaration of the control's background color overrides either type of inheritance. Furthermore, this color inheritance mechanism offers considerable flexibility in displaying components created by legacy applications. When a control must inherit its background color from its parent, either the parent of the original AWT-based object, or that of its Swing counterpart can be used. This option is made possible by the replacement of AWT-based heavyweight Peers by AWTSwing proxy components, discussed earlier. A diagram representing the new color inheritance scheme appears in Fig. 19b.